Tutorial:

Gradescope autograders for programming assignments

Richard Lange (GCCIS)

Thursday, May 15 | 11:00-11:50AM | Wallace Library, Room 3430

Welcome! Please take a seat.

gradescope
by Turnitin

gradescope
by Turnitin

Gradescope autograders at a glance

1. Instructor distributes assignment materials (MyCourses)
2. Instructor uploads autograder.zip to Gradescope
3. Students upload submission.zip to Gradescope
4. In seconds/minutes, students get score + feedback
5. (Instructor/grader patch any autograder issues)
6. Students iterate on solutions until the deadline
7. Instructor/grader put eyes on final submissions

Student perspective on autograders

Pros

- Engagement way up
  - Gamification of assignments
  - Immediate feedback
  - Clear specifications
  - Leaderboards
- Hidden test cases encourage debugging & critical thinking

Cons

- Fixed API, limited creativity
- Causes stress when tests break and they assume it's their fault

Instructor perspective on autograders

Pros

- Student engagement way up
- Automation
- Docs* and support from Gradescope team

Cons

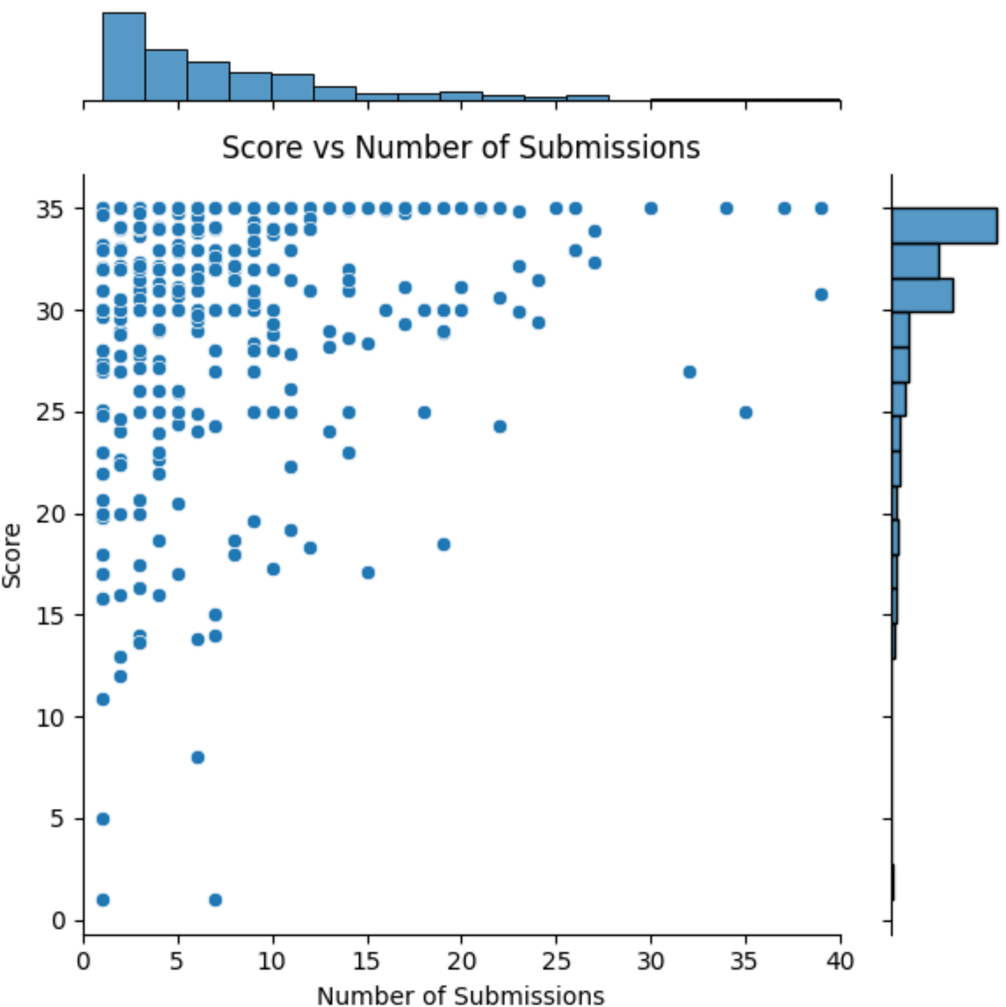
- Can limit assignment design
  - Assignments must be testable
- Provide skeleton code
- Initial investment to get started
  - Hopefully this presentation helps
- Some sharp edges; who tests the tests?

*https://gradescope-autograders.readthedocs.io/en/latest/

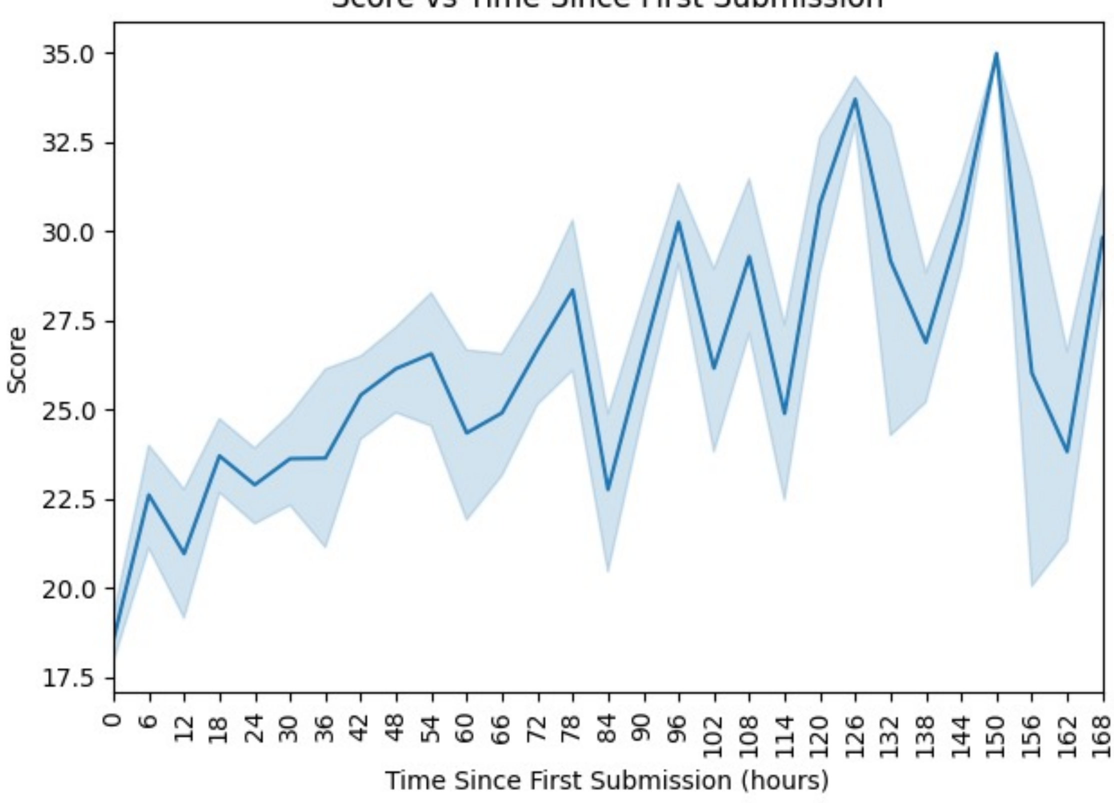
Overall approach

- I divided my course into
  - Programming assignments with clear goals and testable functions to be handled by the autograder
  - Weekly in-class 10-minute quizzes to assess everything else
- Getting started: very usable examples and documentation
  - https://github.com/gradescope/autograder_samples/
  - https://gradescope-autograders.readthedocs.io/en/latest/
- Customizing slowly over time
  - I moved all file-wrangling commands into a Makefile
  - I forked the python `gradescope_utils` package and added some things

Data from CSCI 631



Data from CSCI 631



Demo

Autograders behind the scenes

- Student uploads a zip file (of anything!) to Gradescope
  - In my case, this includes a bunch of python files
- Gradescope spins up a server (AWS, Docker) containing student files and autograder files
- Autograder **must** contain a `run_autograder` shell script, and this script **must** write scores/feedback to a `results.json` file

This is very flexible and customizable: autograder can do anything that a shell script can do, like calling any other tools you can think of.

…but it's easiest to get started with existing tools and templates

Brainstorm

How can you design your assignments for autograding?

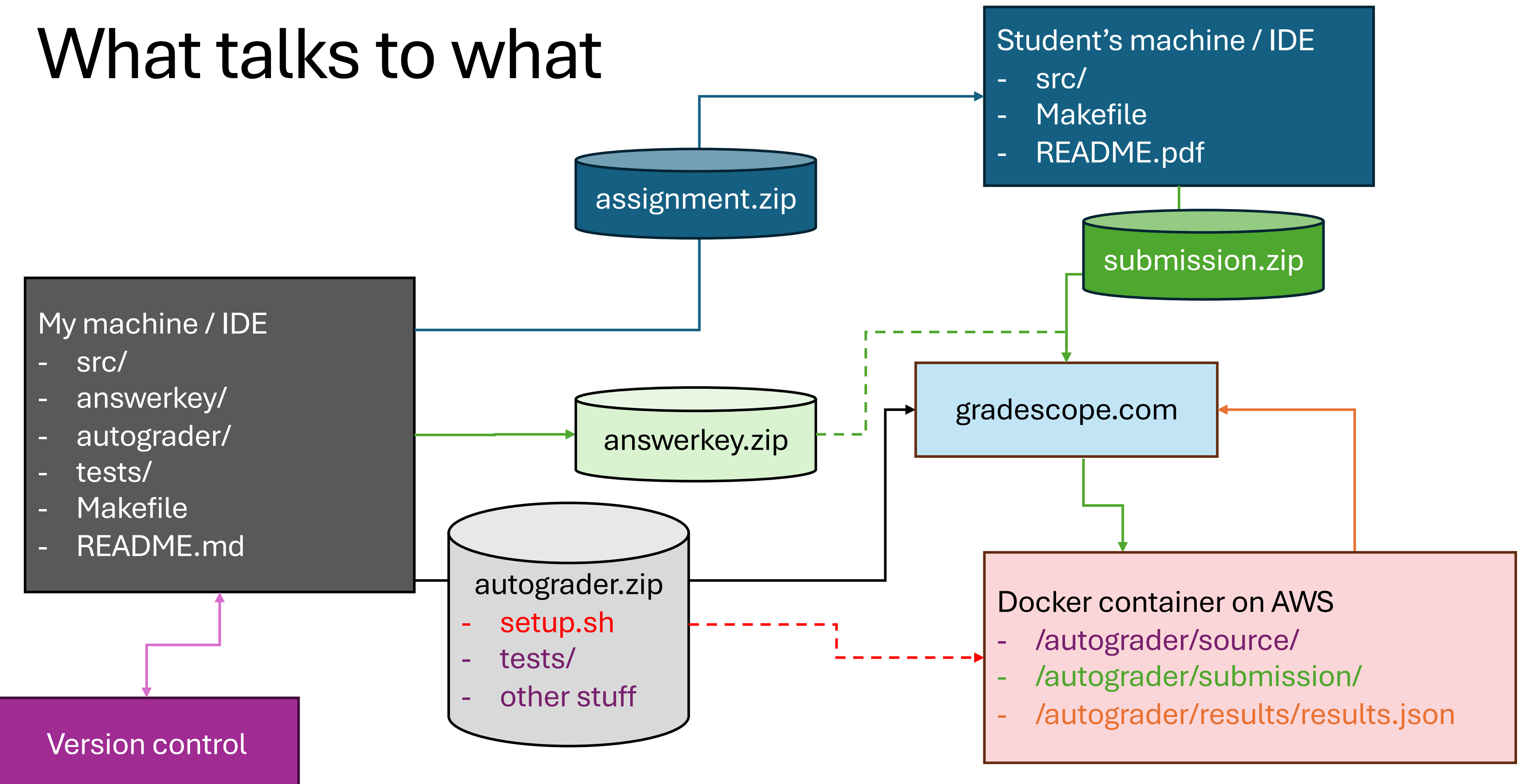
What other syllabus adjustments might you make?

How I create assignments and unit tests using a mix of provided + custom tools

Example from CSCI 631 Foundations of Computer Vision

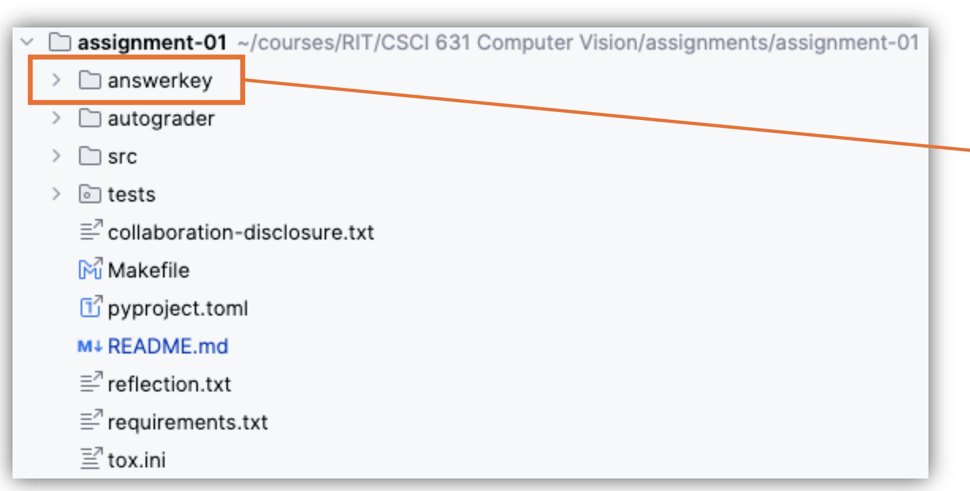
All my assignments are available at https://git.cs.rit.edu/csci631/assignments (ask for access)

What talks to what



https://gradescope-autograders.readthedocs.io/en/latest/specs/

How I create assignments in an IDE (PyCharm)

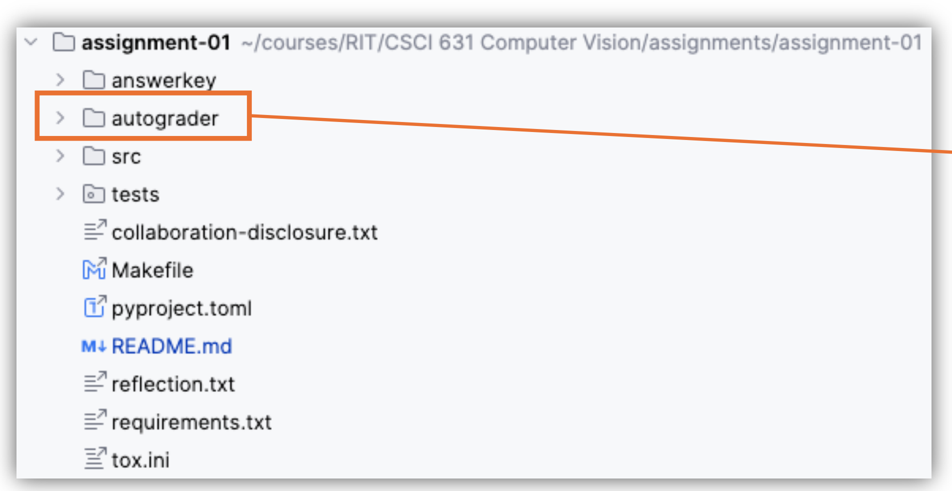


I write this first.
Contains all solutions.
Verify that tests all pass before releasing.

How I create assignments in an IDE (PyCharm)

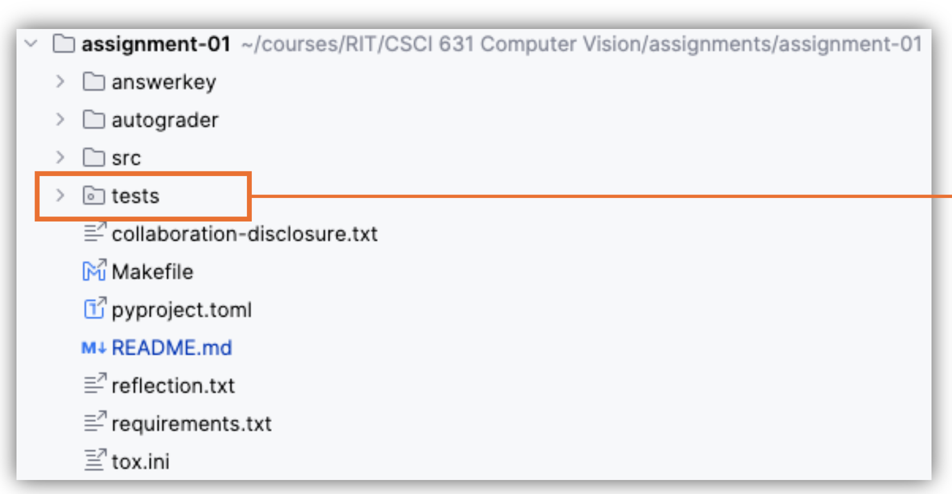
I write this last.
It's a copy of everything in answerkey/ but with lines deleted/modified. The assignment is to write/fix whatever is in src/

How I create assignments in an IDE (PyCharm)



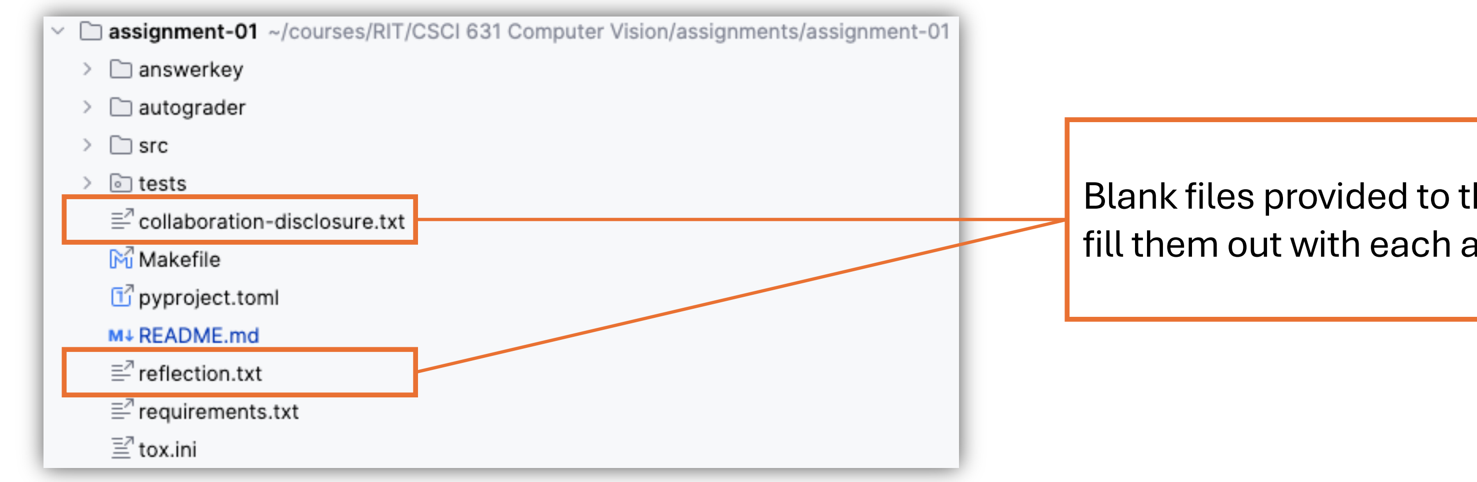
Contains scripts that Gradescope will run on their servers (Dockerfile setup, main script, helpers). Mostly copied from the Gradescope docs. Same for all assignments.

How I create assignments in an IDE (PyCharm)



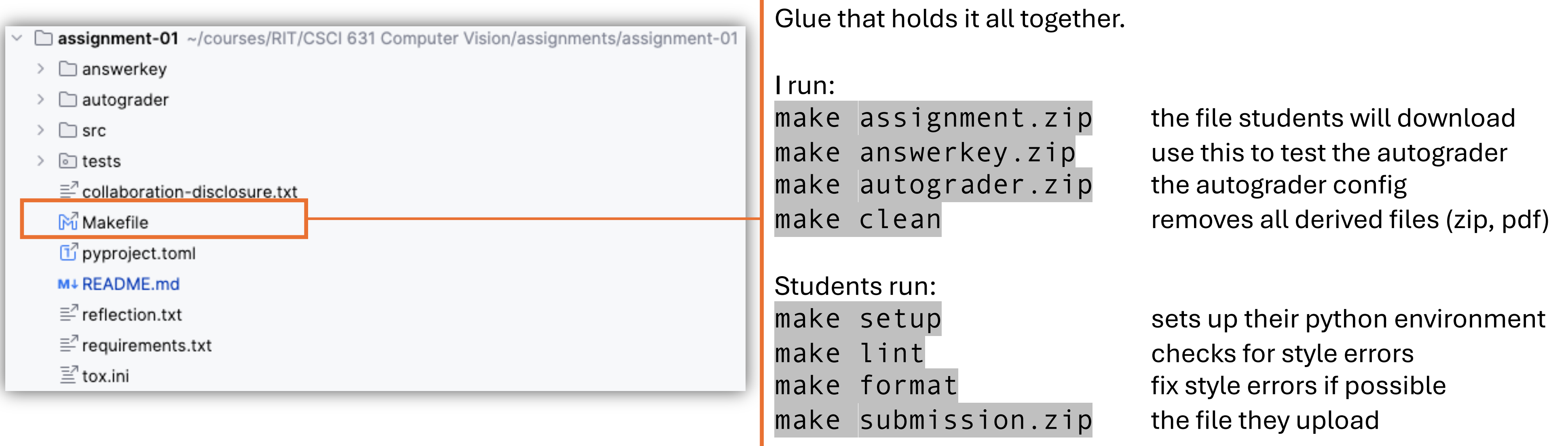
Where the unit tests live.
Students never see the tests themselves, but see point values and error messages.

How I create assignments in an IDE (PyCharm)

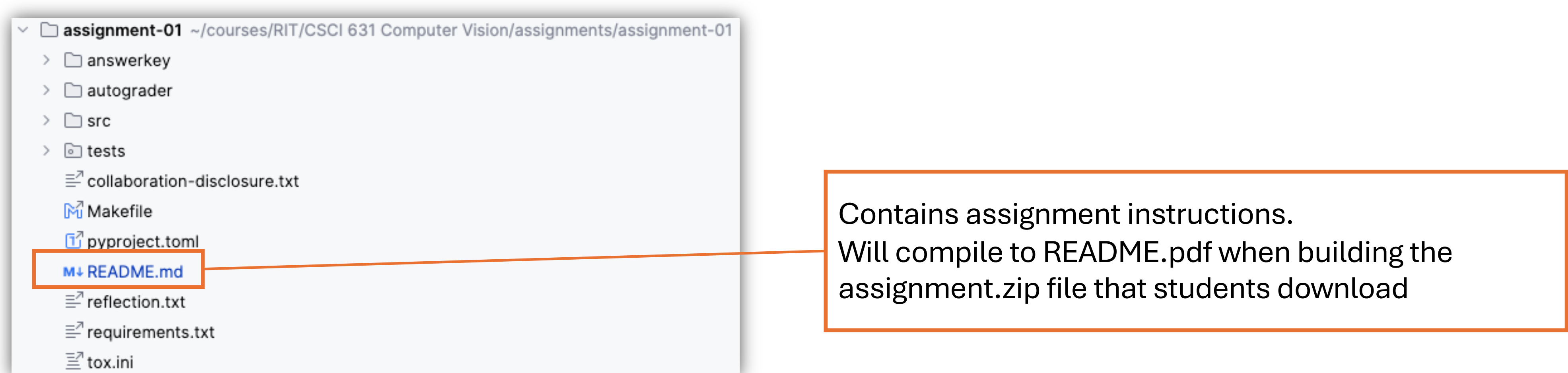


Blank files provided to the students to remind them to fill them out with each assignment.

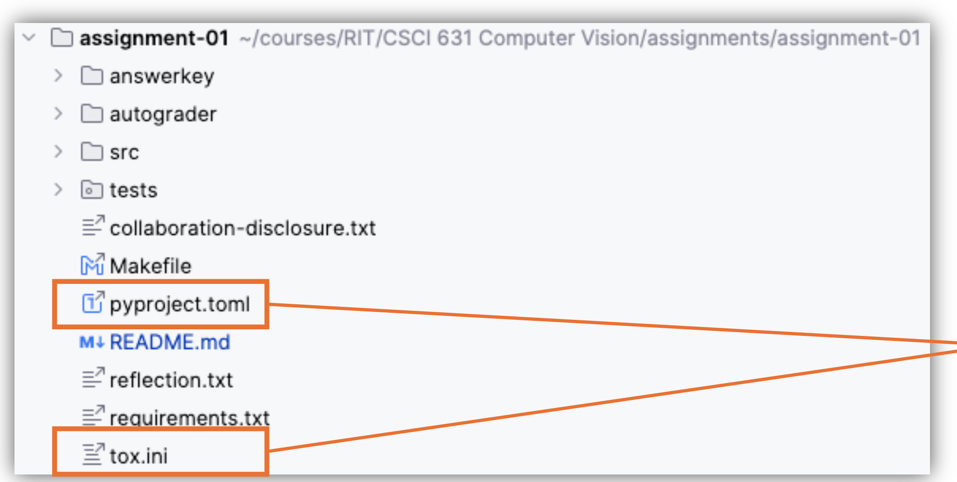
How I create assignments in an IDE (PyCharm)



How I create assignments in an IDE (PyCharm)

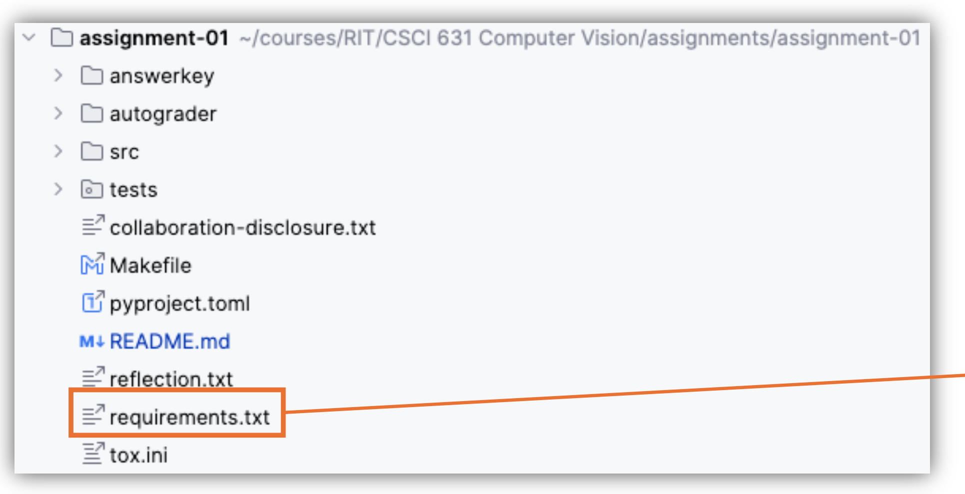


How I create assignments in an IDE (PyCharm)



Configuration files for the linter and formatter.
My autograder deducts up to 5/40 points for style errors.
I allow students to modify tox.ini with their submission;
they must conform to *a* style, not necessarily *my* style

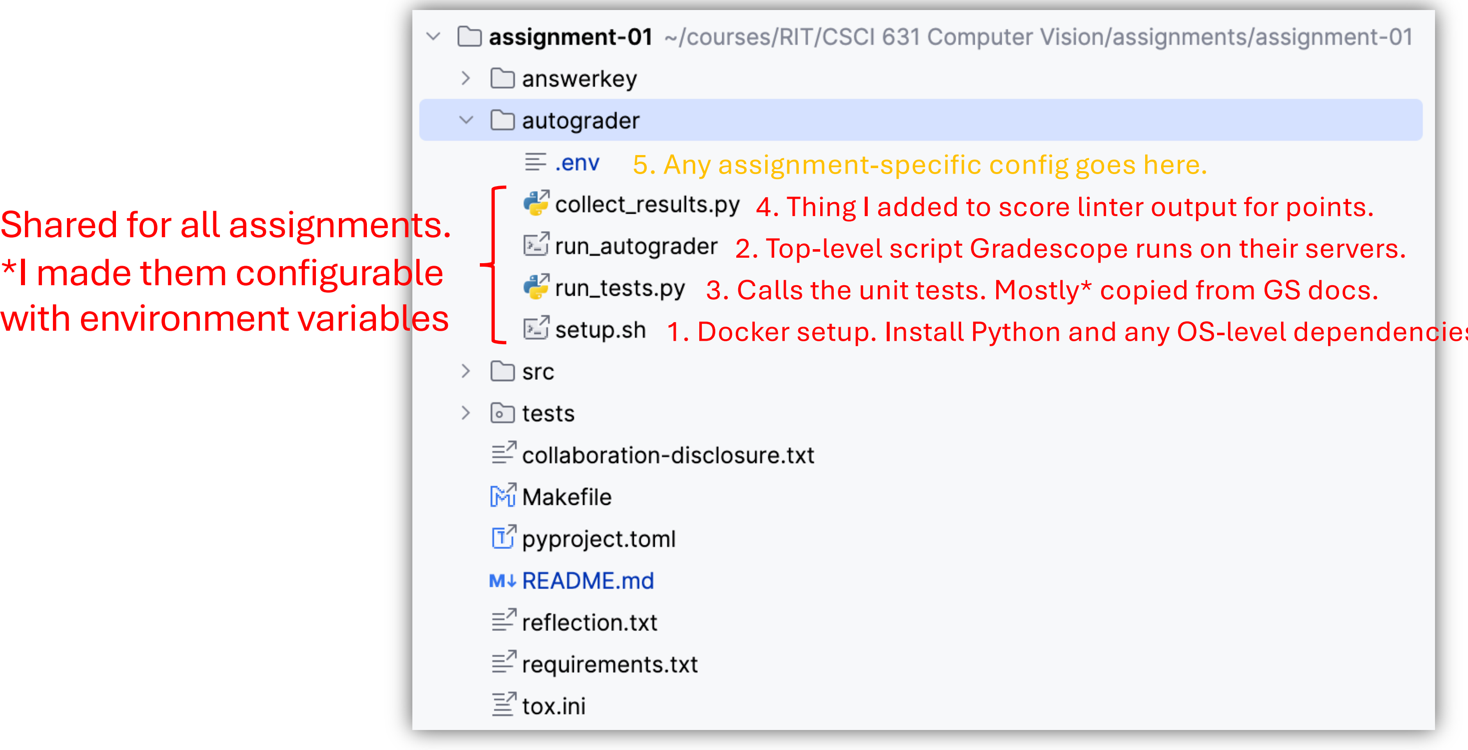
How I create assignments in an IDE (PyCharm)



Python dependencies file. Will be used to set up the students' environments and the autograder environment.

What's in the autograder/ directory

Shared for all assignments.
*I made them configurable
with environment variables



What’s in my setup.sh file

This file is run once when creating the Docker image.
Think of it as “set up a brand-new ubuntu machine to run the student code”

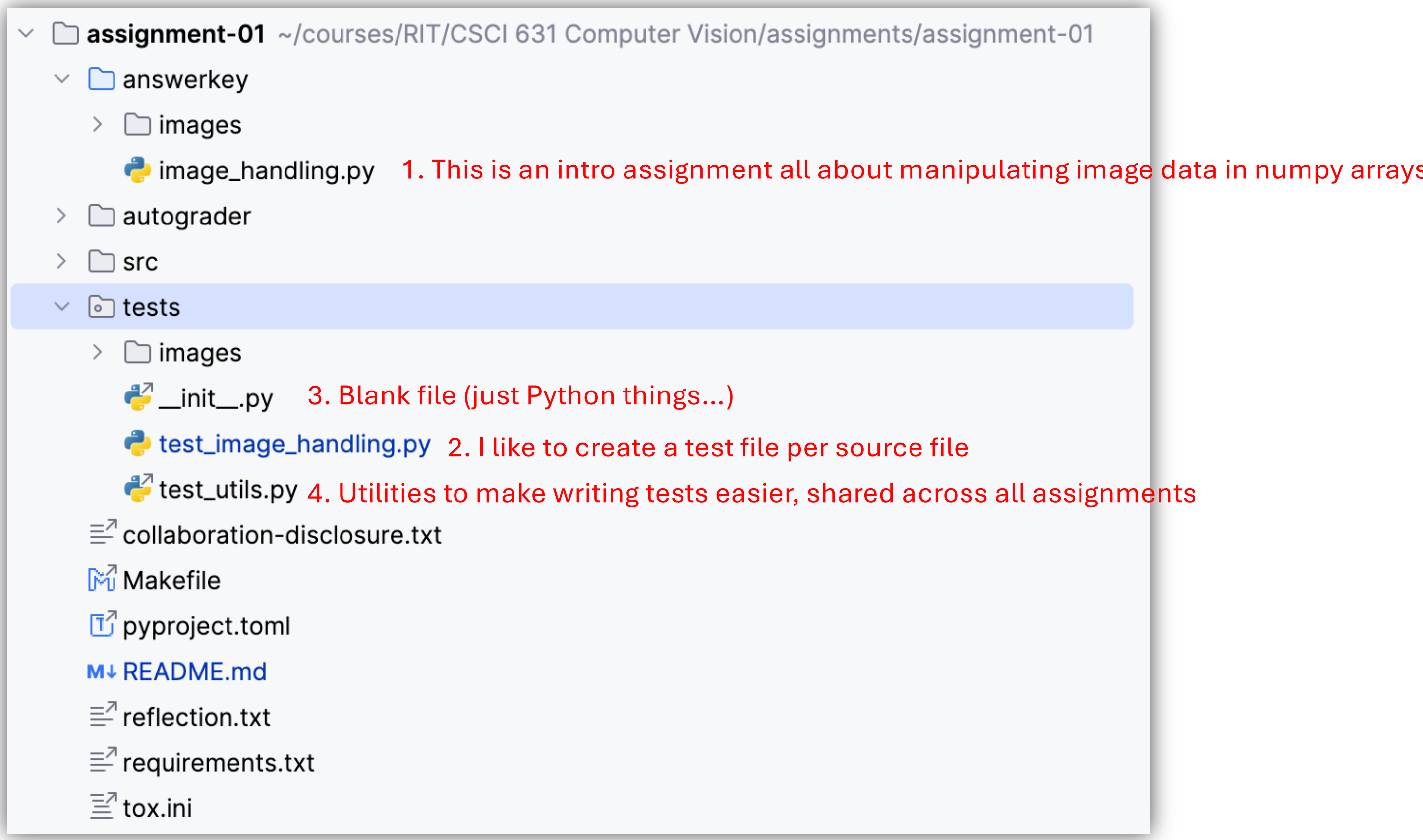
```
apt-get update && apt-get upgrade -y
# Ensure OpenCV dependencies are installed
apt-get install ffmpeg libsm6 libxext6  -y
# Ensure Python 3.10 is installed
apt-get install -y python3.10 python3.10-dev python3.10-venv

# DLib requires CMake
apt-get install -y cmake

# Install Python packages
pip3 install --upgrade pip
pip3 install -r /autograder/source/requirements.txt
pip3 install python-dotenv

# Use my fork of the gradescope-utils package rather than the official one
# (I added/fixed some things)
pip3 install git+https://github.com/wrongu/gradescope-utils.git
```

What’s in my run_autograder file

This is the entry-point for the autograder. GS provides examples.

```
#!/bin/bash

# Move various files from /autograder/source into the student’s submission directory
# so we can run things from there.
cp -r /autograder/source/tests /autograder/submission/src/
cp /autograder/source/run_tests.py /autograder/submission/src/
cp /autograder/source/collect_results.py /autograder/submission/src/
cp /autograder/source/.env /autograder/submission/src/

# Change to the submission/src/ directory
cd /autograder/submission/src/ || exit

# Run flake8 on the student code and collect output. This uses the student’s tox.ini by
default.
flake8 *.py > /autograder/results/flake8.txt

# Run the tests, which generates the initial /autograder/results/results.json file.
python3.10 run_tests.py

# Combine all results into a single results.json file
python3.10 collect_results.py
```

What's in the tests/ directory



What’s in the test_image_handling.py file

```
from gradescope_utils.autograder_utils.decorators import weight, timeout
from tests.test_utils import *
import unittest
from gradescope_utils.autograder_utils.decorators import weight, timeout
from tests.test_utils *


class TestImageHandling(unittest.TestCase):
    ...

    @timeout(1)
    @weight(1)
    def test_grayscale_easy(self):
        """Test image_handling.grayscale function on all 0 or 255 in each channel"""
        from image_handling import grayscale

        assert_no_cv(grayscale)
        bgr = cv.imread("tests/images/bgr.png")
        ref = cv.cvtColor(cv.cvtColor(bgr, cv.COLOR_BGR2GRAY), cv.COLOR_GRAY2BGR)
        assert_ints_close_within_rounding_error(ref, grayscale(bgr))

    ...
```

What's in the test_image_handling.py file

```
from gradescope_utils.autograder_utils.decorators import weight, timeout
from tests.test_utils import *
import unittest
from gradescope_utils.autograder_utils.decorators import weight, timeout
from tests.test_utils *


class TestImageHandling(unittest.TestCase):
    ...

    @timeout(1)
    @weight(1)
    def test_grayscale_randomized(self):
        """Test image_handling.grayscale function on random data"""
        from image_handling import grayscale

        assert_no_cv(grayscale)

        bgr = np.random.randint(0, 256, (100, 100, 3), dtype=np.uint8)
        ref = cv.cvtColor(cv.cvtColor(bgr, cv.COLOR_BGR2GRAY), cv.COLOR_GRAY2BGR)
        assert_ints_close_within_rounding_error(ref, grayscale(bgr))

    ...
```

What’s in the test_image_handling.py file

```python
from gradescope_utils.autograder_utils.decorators import weight, timeout
from tests.test_utils import *
import unittest
from gradescope_utils.autograder_utils.decorators import weight, timeout
from tests.test_utils *


class TestImageHandling(unittest.TestCase):
    ...

    @timeout(1)
    @weight(1)
    def test_grayscale_is_vectorized(self):
        """Check image_handling.grayscale is vectorized (no for or while loops)"""
        from image_handling import grayscale

        assert_no_loops(grayscale)

    ...
```

Tips and tricks

Tip #0: read the docs, use their examples

- Autograder documentation is quite readable
- The gradescope-utils python package is not hard to use either
  - I have a fork where I added some things:
    - @timeout decorator so individual tests fail if stuck in long loops
    - @custom_format decorator so I can flag output type as HTML from my unit tests
    - A rate-limiter preventing students from spamming too many submissions
  - There are packages for other languages… I haven't tried them
- Everything I'm showing today was "start small and build my own tools of the course of a semester"

Trick #1: displaying images back to students

in test_utils.py:

```
from PIL import Image
import base64
from io import BytesIO


def image_to_html(image: Image) -> str:
    buffered = BytesIO()
    image.save(buffered, format="JPEG")
    img_str = base64.b64encode(buffered.getvalue()).decode("utf-8")
    return f'<img src="data:image/jpeg;base64,{img_str}" />'
```

Warning: if images are too big, this string gets truncated by GS and won't display properly.

in test_student_code.py:

```
@weight(1 / 3)
@custom_output(format="html", mode=None)
def test_blue(self):
    """Test that generated image is correct"""
    from student_code import generate_image

    im = generate_image()
    ref = cv.imread("tests/images/reference.png")
    assert np.allclose(ref,im), \
        f"Image mismatch! Expected: {image_to_html(ref)} Actual: {image_to_html(im)}"
```

← This is a custom decorator I made

Tip #2: many small tests

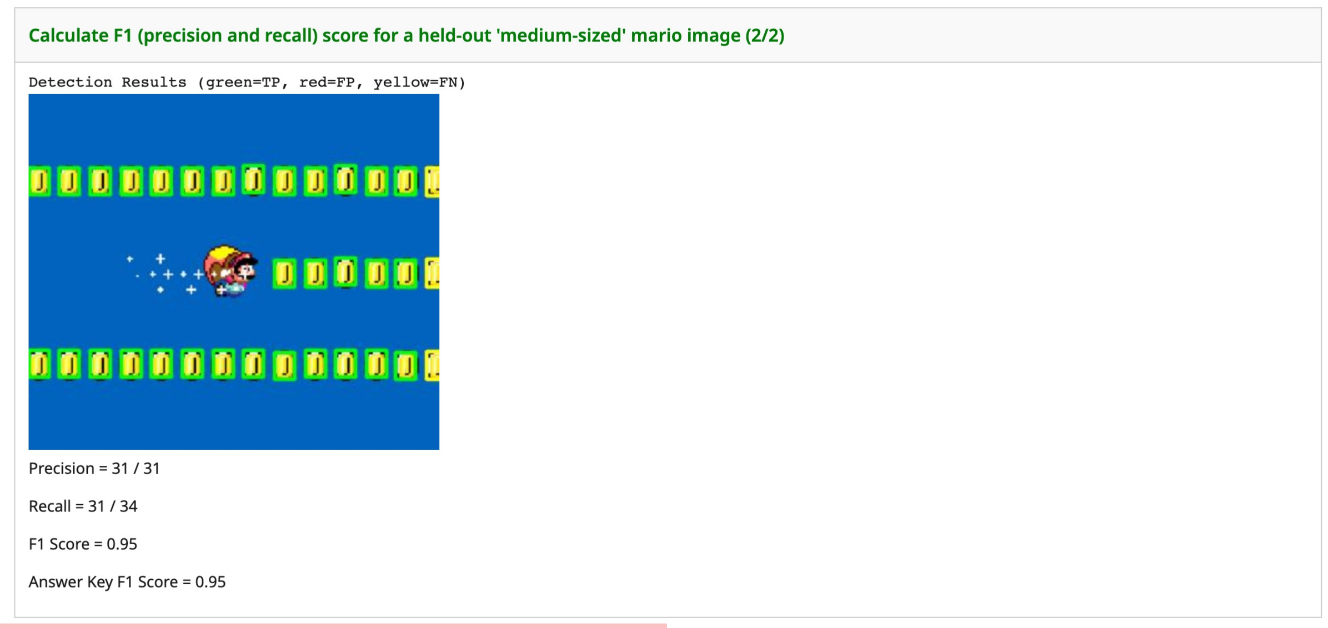
Instead of this:

```
@partial_credit(2)
def test_blue(self, set_score=None):
    """Testing a few cases"""
    if case1:
        set_score(1)
    if case2:
        set_score(2)
```

I recommend this:

```
@weight(1)
def test_case1(self):
   """Testing some specific case 1"""
    assert case1, "Informative message about case1"

@weight(1)
def test_case2(self):
    """Testing some specific case 2"""
    assert case2, "Informative message about case2"
```

Tip #3: test the behavior, not the output

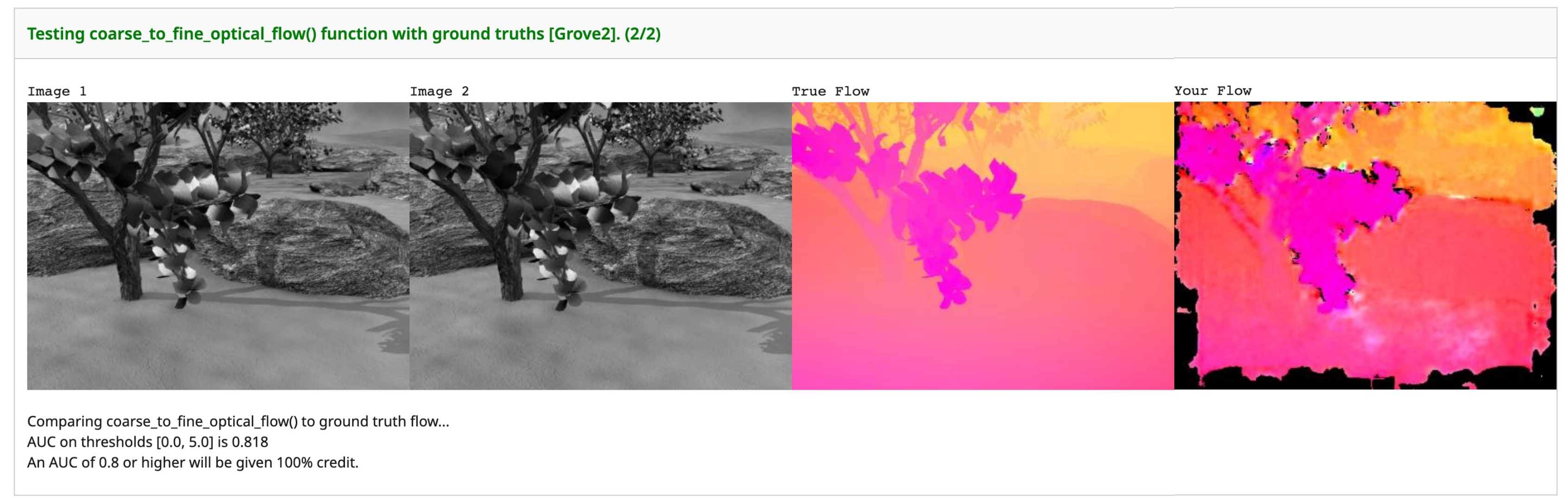


BAD:
Assert exact match to answer key

BETTER:
- assert precision and recall > threshold like 0.9
- or, use @partial_credit to give scores

BEST:
- assert underlying functions work individually
- create leaderboard for who can do best

Tip #3: test the behavior, not the output

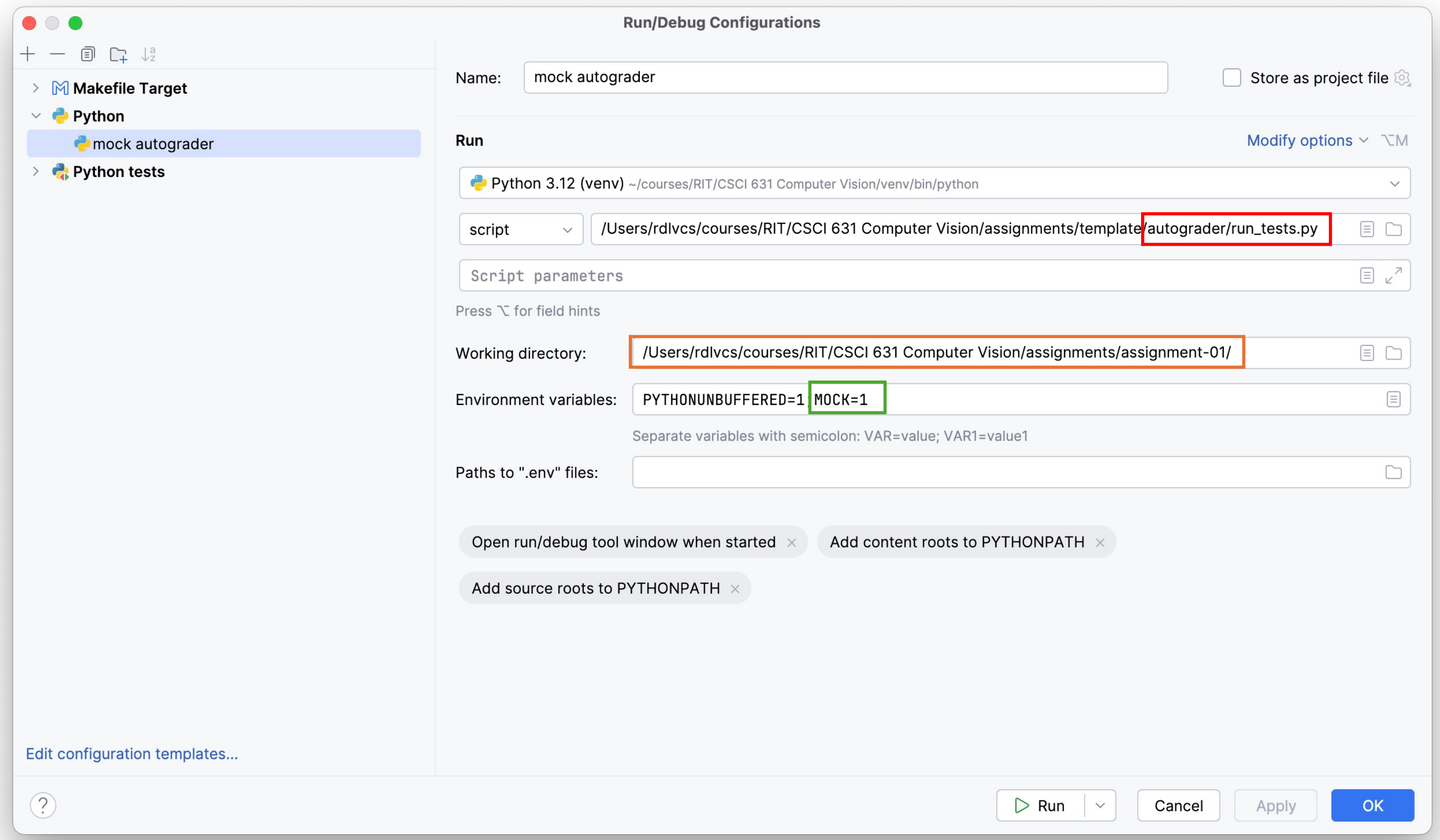


BAD:
Assert exact match to answer key

BETTER:
- assert precision and recall > threshold like 0.9
- or, use @partial_credit to give scores

BEST:
- assert underlying functions work individually
- create leaderboard for who can do best

Trick #4: local debugging without local docker



This script discovers unit-tests in tests/, runs them, and populates results.json

Tell PyCharm to run from assignment root

I edited run_tests.py so that when MOCK is set, saves to "results.json" rather than "/autograder/results/results.json"

Not pictured: one of either answerkey/ or src/ is configured as "sources root" depending on which I'm testing

Summary

- Autograder benefits:
  - **Student engagement.** Gamification and opportunity to try and retry
  - **Automation.** Long-term time-saving. Equal treatment of all students.
- Autograder sharp edges:
  - **Learning curve.** Takes investment to get started.
  - **Live patching.** It is hard to write tests that pass for all correct submissions and none of the incorrect ones
  - **Rigidity**. I would love to figure out how to allow creative problem-solving / high-level design

Questions?

Reach out: rdlvcs@rit.edu